Detailed Project Plan

A Comprehensive Pure-Haskell Native Desktop Reflex-based Functional Reactive Programming GUI Framework + Example Application

Lai Yat Kit
University ID: 3036140179

Dr. Bruno Oliveira
Project Supervisor

September 29, 2024

Contents

1 Background

Haskell is one of the most well-known general-purpose purely functional programming languages. To develop native desktop graphical user interfaces (GUIs) in Haskell, it is often necessary to utilize third-party libraries. There are several open-source Haskell libraries created for this purpose, each offering different GUI widget toolkits and employing different GUI programming paradigms. Programmers shall freely pick one that satisfies their needs. However, if one hopes to gain from the numerous benefits of designing software with functional programming principles [1, 2, 3], they might prefer to use a Haskell native desktop GUI library equipped with functional reactive programming (FRP) as its GUI programming paradigm, complete with a widget toolkit built in pure-Haskell, as opposed to using a typical Haskell library that employs an imperative programming paradigm and is built on top of an external widget toolkit, such as GTK or Qt. Yet currently, such a Haskell library does not exist.

The following sections will further elaborate on why such a library could be highly desirable, and highlight its advantages over libraries using a foreign widget toolkit or employing a different GUI programming paradigm.

1.1 A library should have a pure-Haskell built GUI widget toolkit

Regarding GUI widget toolkits, an existing Haskell GUI library could either provide a foreign widget toolkit, such as Qt[1] or GTK[2], or offer a widget toolkit built purely in Haskell. Currently, most Haskell GUI libraries depend on a foreign widget toolkit, with notable examples being libraries `gi-gtk`, `gi-gtk-declarative`, and `reflex-gi-gtk`, which are based on GTK. However, there exists unique technical challenges of integrating Haskell with foreign GUI widget toolkits. It is not trivial to expand upon foreign GUI widget toolkits due to limitations of the Haskell FFI[3]. For instance, it is not possible to create new GTK widgets in pure-Haskell using any of the previously mentioned Haskell libraries that are based on GTK. The issue typically stems from the reliance on C++ inheritance and C ABI specifications by most established foreign GUI widget tool, which Haskell FFI has limited control of. This technical constraint strongly motivates the need for a GUI widget toolkit built in pure-Haskell, as such a toolkit naturally exposes first-class programming interfaces to enable an idiomatic and seamless Haskell GUI development experience.

As of now, it appears that the Haskell library `monomer` is one of the few, if not the only, Haskell native desktop GUI library employing a pure-Haskell GUI widget toolkit.

1.2 A library should employ a declarative programming paradigm

Building on the benefits of a pure-Haskell GUI widget toolkit, a library should also employ a declarative programming paradigm instead of a traditional imperative programming paradigm. The advantages of declarative programming paradigms could be highlighted by the ease of using it to design maintainable event-driven software when compared to traditional imperative styles. Native desktop GUI applications often involve handling multiple nonsynchronous tasks simultaneously, such as responding to user input, updating graphics, and managing asynchronous events. Building applications with common traditional imperative programming techniques, such as callbacks, often leads to unmaintainable and complex codebases if the application has complex software requirements [4]. In contrast, declarative programming paradigms provide a more elegant solution. By defining the desired outcome in high-level and delegating low-level logics to the underlying GUI library, developers could write concise and readable code that could evolve with ease. In light of this, one might prefer to use a Haskell native desktop GUI library that employs a declarative programming paradigm.

In the context of Haskell, two prominent declarative programming paradigms for GUI development are The Elm Architecture and functional reactive programming. The Elm Architecture, pioneered by the programming language Elm, offers a simple and intuitive approach to structuring and designing GUI programs. Conversely, functional reactive programming is more challenging to learn and utilize, yet it was argued that FRP enables the creation of more comprehensible codebases, as discussed in [5]. Therefore, it could be seen that both paradigms have their respective merits. A developer might opt a library with either one of these two options.

1.3 The problem

While there exists a Haskell library offering a pure-Haskell GUI widget toolkit framework with The Elm Architecture as its declarative programming paradigm, namely `monomer`, there currently

[1]https://www.qt.io/

[2]https://www.gtk.org/

[3]Haskell FFI documentation: https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/ffi.html

exists no alternative that employs functional reactive programming. As discussed above, such an alternative could be highly desirable.

2 Objective

The primary goal of this project is to develop a new Haskell library named `RGX`, a native desktop GUI library with a pure-Haskell GUI framework and functional reactive programming, enabled through the Haskell FRP library `reflex`. The project aims to equip `RGX` with a wide range of GUI features, striving to deliver functionalities that is comparable to well-established GUI frameworks such as GTK and Qt, as well as providing thorough documentations to allow new library users to quickly understand its usage. Within the scope of this final year project, `RGX` shall include, but not limited to, the following features and functionalities:

Basic GUI widgets

`RGX` shall provide common GUI widgets including, but not limited to, text labels, pictures, push buttons, checkboxes, radio buttons, text fields, scroll bars, scroll view, lists, tree lists, dropdowns, and combo boxes.

Widget layout utilities

`RGX` shall provide utilities to layout a set of widgets logically. For example, in Qt, there is `QHBoxLayout` and `QVBoxLayout` to layout a list of widgets horizontally and vertically respectively. `RGX` shall offer such utilities too.

Styling and theming

`RGX` shall provide utilities to style the visual appearances of widgets. Programmers shall be able to style widget individually, as well as defining a theme.

Animations

`RGX` shall provide utilities to animate widget visuals. For example, in HTML/CSS, it is possible to write CSS to animate the background of a `<button>` when the user's cursor is hovering above the widget. `RGX` shall offer such utilities too, possibly through a different programming method than that of the example.

Multi-window

`RGX` shall provide mechanisms to allow applications to create new sub-windows and manage them.

Pop-up dialogs

`RGX` shall provide utilities to create pop-up dialogs, a common utility found in many modern GUI frameworks.

Audio playback

`RGX` shall provide utilities to play simple audio cues. For example, it shall be possible to create a push button that, when clicked, plays a user-defined audio cue.

Low-level extensions

`RGX` shall provide mechanisms to allow developers to easily extend the GUI library. For instance, developers shall be able to create highly-optimized GUI widgets using unsafe I/O code and direct OpenGL calls. This feature could be crucial for building widgets such as a data graph widget that can handle a substantial volume of data or a high-performance video player widget that requires unsafe I/O interoperability with GStreamer[4] for CPU/GPU intensive multimedia processing.

In addition to developing `RGX`, this project will also showcase an example use of the library by developing a GUI application named `RGXSyncthing`, a frontend for Syncthing[5]. This example has been chosen specifically because of its moderately complex software requirements, particularly in the need to effectively manage asynchronous network requests and GUI reactivity, which creates an interesting challenge for `RGX`.

3 Methodology

For the general software development of `RGX` and `RGXSyncthing`, the project will use Git as the version control system. The development environment is managed with Nix[6] flakes, which pin third-party library versions to ensure reproducibility. Cabal[7] is used for structuring the project's Haskell packages and managing dependencies, while haddock[8] generates documentations from comments in the project's Haskell source code.

To implement `RGX`, a number of dependencies and APIs will be used. For functional reactive programming, the Haskell library `reflex`[9] will be used as the underlying functional reactive programming library. Even though there exists alternative FRP libraries, this project specifically chose `reflex` due to its widespread adoption in the industry, as observed in the Haskell community. In terms of implementing graphics, audio, and the windowing system, `RGX` relies on SDL[10] to interface with the execution environment's operating system and hardware to achieve that, while OpenGL[11], FreeType[12], and HarfBuzz[13] are used to render widgets and text elements.

To implement `RGXSyncthing`, the only major graphical library dependency used is `RGX`, as this is to demonstrate the capabilities of `RGX`. However, capabilities such as networking, which is vital in the Syncthing protocol, will be supported through other open-source Haskell libraries. The precise details shall be determined once `RGXSyncthing` begins development after `RGX` is completed.

4 Schedule and Milestones

The following is a list of project milestones and their definitions:

[4] https://gstreamer.freedesktop.org/
[5] https://syncthing.net/
[6] https://nixos.org/guides/how-nix-works/
[7] https://www.haskell.org/cabal/
[8] https://hackage.haskell.org/package/haddock
[9] https://github.com/reflex-frp/reflex
[10] https://www.libsdl.org/
[11] https://www.opengl.org/
[12] http://freetype.org/
[13] https://github.com/harfbuzz/harfbuzz

- **Milestone A10**

  An initial version of `RGX` has been developed. The foundation of `RGX` is now sufficient to bootstrap the development of `RGXSyncthing`.

- **Milestone A20**

  `RGX` has been fully developed and is ready for submission. The library must also be fully documented with Haddock.

- **Milestone B10**

  The example application `RGXSyncthing` has been completed and is ready for submission.

The following is the tentative schedule of the project:

Week	Date	Tasks
		First semester begins
1	01/09/24	Setup and configure project
2	08/09/24	
3	15/09/24	
4	22/09/24	
5	29/09/24	Phase 1 (Inception)
6	06/10/24	
		Reading week begins
7	13/10/24	
		Reading week ends
8	20/10/24	
9	27/10/24	Achieve **Milestone A10**
		Begin researching the Syncthing protocol
		Begin development of `RGXSyncthing`
10	03/11/24	
11	10/11/24	
12	17/11/24	
13	24/11/24	Achieve **Milestone B10**
		Halt project due to upcoming examination
		Revision period begins
14	01/12/24	
		Assessment period begins
15	08/12/24	
16	15/12/24	
17	22/12/24	Resume project after examination
		Assessment period ends
18	29/12/24	Achieve **Milestone A20**
19	05/01/25	
20	12/01/25	First Presentation
		Second semester begins

21	19/01/25	
22	26/01/25	Phase 2 (Elaboration)
23	02/02/25	
24	09/02/25	
25	16/02/25	
26	23/02/25	
27	02/03/25	
Reading week begins		
28	09/03/25	
Reading week ends		
29	16/03/25	
30	23/03/25	
31	30/03/25	Phase 3 (Construction)
32	06/04/25	
33	13/04/25	
34	20/04/25	
35	27/04/25	Final Presentation
Project ends		

References

[1] B. Ray, D. Posnett, P. Devanbu, and V. Filkov, "A large-scale study of programming languages and code quality in GitHub," *Commun. ACM*, vol. 60, no. 10, p. 91–100, Sep. 2017. [Online]. Available: https://doi.org/10.1145/3126905

[2] E. D. Berger, C. Hollenbeck, P. Maj, O. Vitek, and J. Vitek, "On the impact of programming languages on code quality: A reproduction study," *ACM Trans. Program. Lang. Syst.*, vol. 41, no. 4, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3340571

[3] P. Hudak and M. P. Jones, "Haskell vs. Ada vs. C++ vs. awk vs.... an experiment in software prototyping productivity," *Contract*, vol. 14, no. 92-C, p. 0153, 1994.

[4] K. Gallaba, A. Mesbah, and I. Beschastnikh, "Don't call us, we'll call you: Characterizing callbacks in javascript," in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2015, pp. 1–10.

[5] S. Krouse, "Explicitly comprehensible functional reactive programming," *REBLS'18*, 2018.